# Chapter 12

## Optimal Instruction Set

In the preceding chapters, we uncovered the following shortcomings of the standard instruction set of H1:

- The amount of main memory is insufficient (see Section 2.2).
- Strings are stored inefficiently in memory (see Section 3.11).
- Immediate instructions that add and subtract are lacking (see Section 4.5).
- Index registers are lacking (see Section 4.10).
- There are too few accumulator-type registers (see Section 4.10).
- The `swap` instruction corrupts the **sp** register (see Sections 4.4 and 7.9).
- Multiply and divide instructions are lacking (see Section 7.2).
- The dual use of the **sp** register as a stack pointer and as a base register for the local instructions causes complications (see Sections 4.9 and 7.5).
- Obtaining the address of a variable located on the stack is difficult (see Sections 7.9 and 9.2).
- A block copy instruction is lacking (see Section 8.4).
- Calling a function given its address at run time is difficult (see Section 8.6).
- The `aloc` and `dloc` instructions are limited (see Section 8.7.1).
- Performing signed and unsigned comparisons are difficult (see Section 8.9).
- Multi-word arithmetic is difficult (see Section 8.10).
- Bit-level operations are not supported (see Section 8.11).

# Constraints on new instructions

1. The hardware, itself, imposes certain constraints. For example, if there are no available registers at the microlevel, we cannot introduce new registers at the machine level.
2. Microstore is a fixed size. On H1, it consists of 512 words. The size of our new microcode cannot exceed this limit.
3. Many opcodes are already taken by existing machine instructions. Figure 12.1 shows all the instructions in the standard instruction set and their opcodes. None of the 4-bit opcodes are available, seven 8-bit opcodes are available, all 12-bit opcodes are available, and five 16-bit opcodes are available.
4. The new microcode should support all the instructions in the standard instruction set. Then any machine program that runs with the old microcode will also run with the new microcode. Any modification to a computer with this property is said to be *backward compatible*. Although not a necessity, backward compatibility is highly desirable. Unfortunately, the standard instruction set has one

# Opcodes in use

**FIGURE 12.1**

| 4-bit opcodes | | 8-bit opcodes | | 12-bit opcodes | 16-bit opcodes | |
|---|---|---|---|---|---|---|
| 0 | ld | F0 | ret | None | FFF5 | uout |
| 1 | st | F1 | ldi | | FFF6 | sin |
| 2 | add | F2 | sti | | FFF7 | sout |
| 3 | sub | F3 | push | | FFF8 | hin |
| 4 | ldr | F4 | pop | | FFF9 | hout |
| 5 | str | F5 | aloc | | FFFA | ain |
| 6 | addr | F6 | dloc | | FFFB | aout |
| 7 | subr | F7 | swap | | FFFC | din |
| 8 | ldc | | | | FFFD | dout |
| 9 | ja | | | | FFFE | bkpt |
| A | jzop | | | | FFFF | halt |
| B | jn | | | | | |
| C | jz | | | | | |
| D | jnz | | | | | |
| E | call | | | | | |

### 12.2.1 mult (Hardware Multiply), m (Shift-add Multiply), div (Divide), and rem (Remainder)

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| FF3 | m | Shift-add Multiply | ac = mem[sp++] * ac; |
| FF4 | mult | Hardware multiply | ac = mem[sp++] * ac; |
| FF5 | div | Divide | if (ac == 0) ac = sp; else ac = mem[sp++] / ac; |
| FF6 | rem | Remainder | if (ac == 0) ac = ct; else ac = mem[sp++] % ac; |

# The mult instruction uses the *stack-ac* approach

- The multiply instruction could use the stack exclusively: It would pop two numbers from the top of the stack, multiply them, and then push the product onto the stack. We call this the *stack-exclusive approach*.
- The multiply instruction could use both the **ac** register and the stack: It would multiply the number in the **ac** register by the number it pops from the stack, and then place the product into the **ac** register. Let's call this the *stack*-**ac** approach.

# Description of mult

```
ac = mem[sp++] * ac;
```

accesses the top of the stack and then increments sp (i.e., pops the stack)

# Computation of y times x

```
ld    y      ; get one number
push         ; push this number onto stack
ld    x      ; load the other number into the ac register
mult         ; multiply the two numbers; product goes into ac
st    z      ; save product
```

# 9 divided by 2

```
ldc  9
push          ; push dividend
ldc  2        ; load ac with divisor
div           ; quotient is placed in the ac register
```

# Add instructions set carry register

## 12.2.2 addc (Add Constant) and subc (Subtract Constant)

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| F8 | addc y | Add constant | ac = ac + y; cy = carry; |
| F9 | subc y | Subtract constant | ac = ac - y; |

### 12.2.3 scmp (Signed Compare)

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| FD | scmp | Signed compare | temp = mem[sp++];<br>if (temp < ac) ac = -1;<br>if (temp == ac) ac = 0;<br>if (temp > ac) ac = 1; |

# What scmp does

1. Subtract B from A. If the result is 0, then B equals A, and we are done.
2. If A and B have like signs, we can use the sign-only test, and we are done. The sign-only test works in this case because overflow never occurs when subtracting numbers with like signs.
3. We reach this step only if A and B have unlike signs. Thus, if A is negative, then B must be positive, implying A is less than B. If A is positive, then B must be negative, implying A is greater than B.

# Using scmp

Code for

```
if (x < y)
    z = 5;
```

is

```
      ld    x
      push
      ld    y
      scmp          ; ac loaded with -1 if x < y
      jzop @L1      ; jump over body if ac 0 or 1
      ldc   5       ; z = 5;
      st    z
@L1:
```

### 12.2.4 ucmp (Unsigned Compare)

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| FE | ucmp | Unsigned compare | Same as signed compare, except with unsigned numbers |

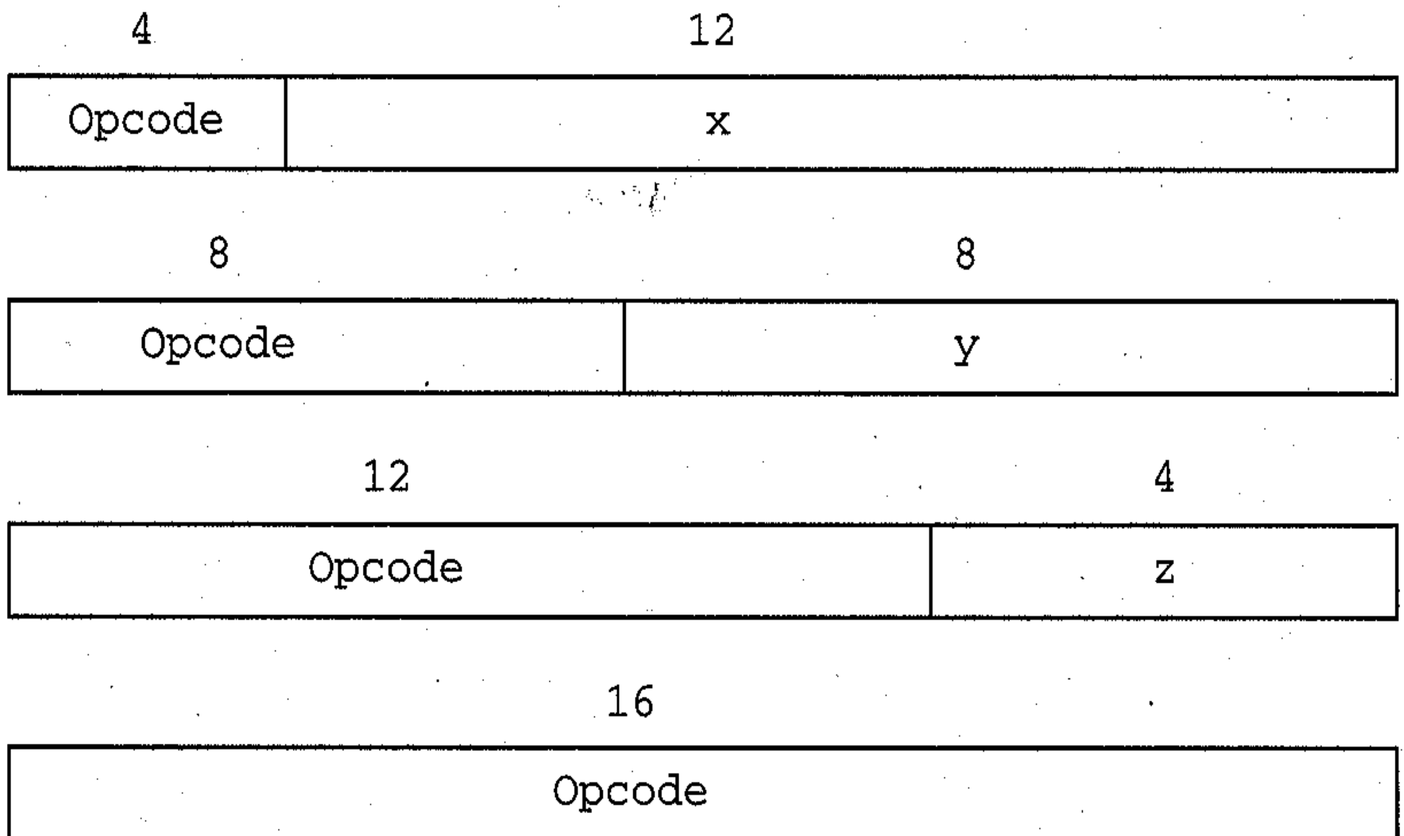
# Logical shift shifts in zeros

## 12.2.5 `shll` (Shift Left Logical) and `shrl` (Shift Right Logical)

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| FF0 | shll z | Shift left logical | ac = ac << z; |
| FF1 | shrl z | Shift right logical | ac = ac >> z; (inject 0's) |

# 4, 8, 12, and 16 bit opcodes

FIGURE 12.2



# x, y, and z fields

| | | |
|---|---|---|
| 4-bit opcode | 12-bit x field | (like `ld`) |
| 8-bit opcode | 8-bit y field | (like `dloc`) |
| 12-bit opcode | 4-bit z field | (like `shll`) |
| 16-bit opcode | | (like `halt`) |

# Using shll to test a single bit

```
shll 2      ; move the third bit from left into sign position
jn   got_one
ja   got_zero
```

Arithmetic shift shifts in the sign bit. It divides positive and negative numbers by 2 for each shift.

### 12.2.6 shra (Shift Right Arithmetic)

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| FF2 | shra z | Shift right arith | ac = ac >> z; (inject sign) |

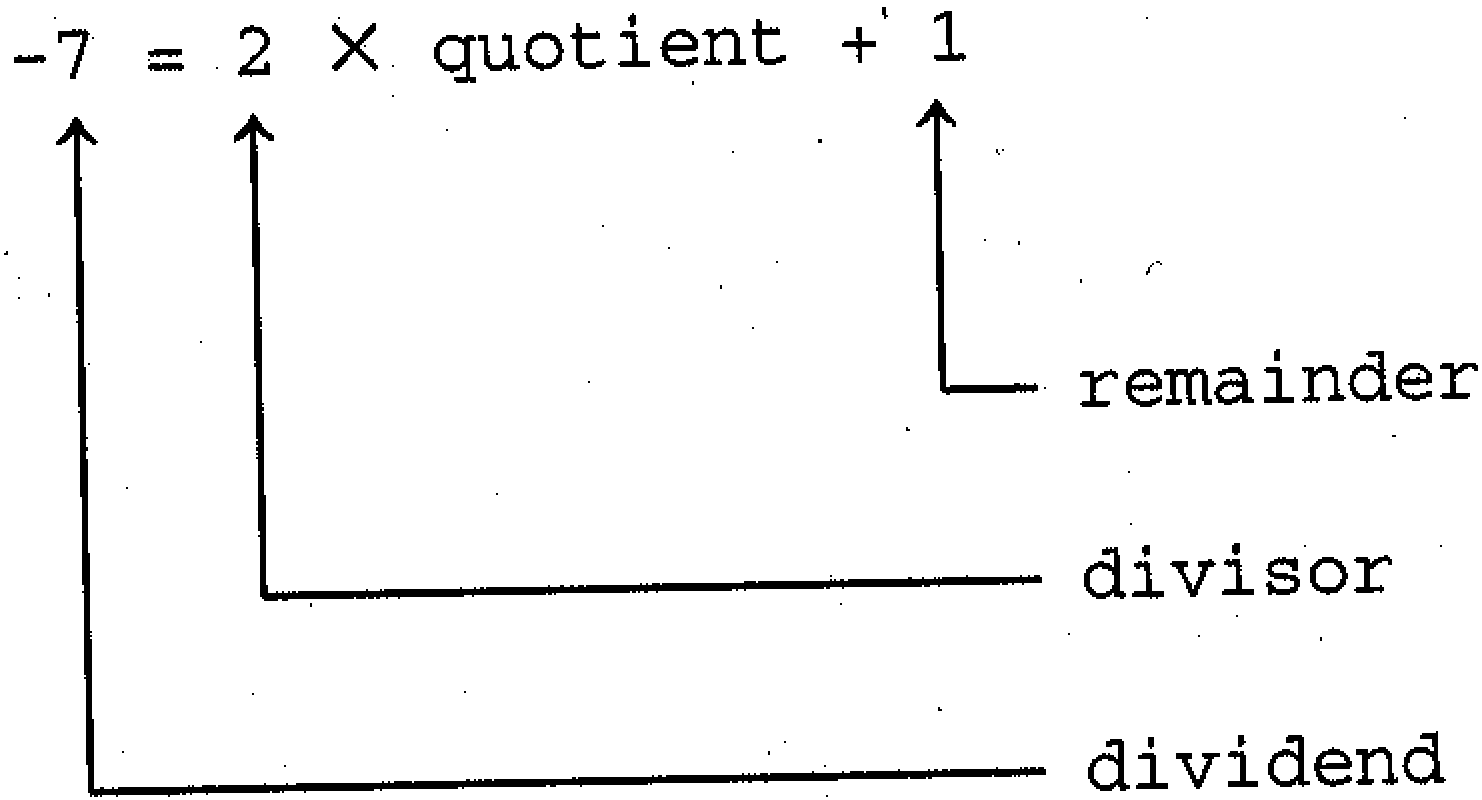
# Definition of integer division

$$\text{dividend} = \text{divisor} \times \text{quotient} + \text{remainder}$$

where $0 \le \text{remainder} < \text{divisor}$.

# Dividing -7 by 2

The shra instruction computes this quotient.



Do Java and C++ yield the quotient and remainder as on the preceding slide? Let's see.

**FIGURE 12.3**

```
class Div {
 public static void main(String arg[])
 {
   System.out.println("Dividend Divisor Quo Rem");
   System.out.println("    7        2      " +  7/2  + "   "  + 7%2);
   System.out.println("   -7        2    "  + -7/2  + " "    + -7%2);
   System.out.println("    7       -2    "  +  7/-2 + "   "  + 7%-2);
   System.out.println("   -7       -2      " + -7/-2 + " "    + -7%-2);
 }
}
```

-7 divided by 2 yields a different quotient and remainder.  The sign of the remainder always is equal to the sign of the dividend.

| Dividend | Divisor | Quo | Rem |
|---|---|---|---|
| 7 | 2 | 3 | 1 |
| −7 | 2 | −3 | −1 |
| 7 | −2 | −3 | 1 |
| −7 | −2 | 3 | −1 |

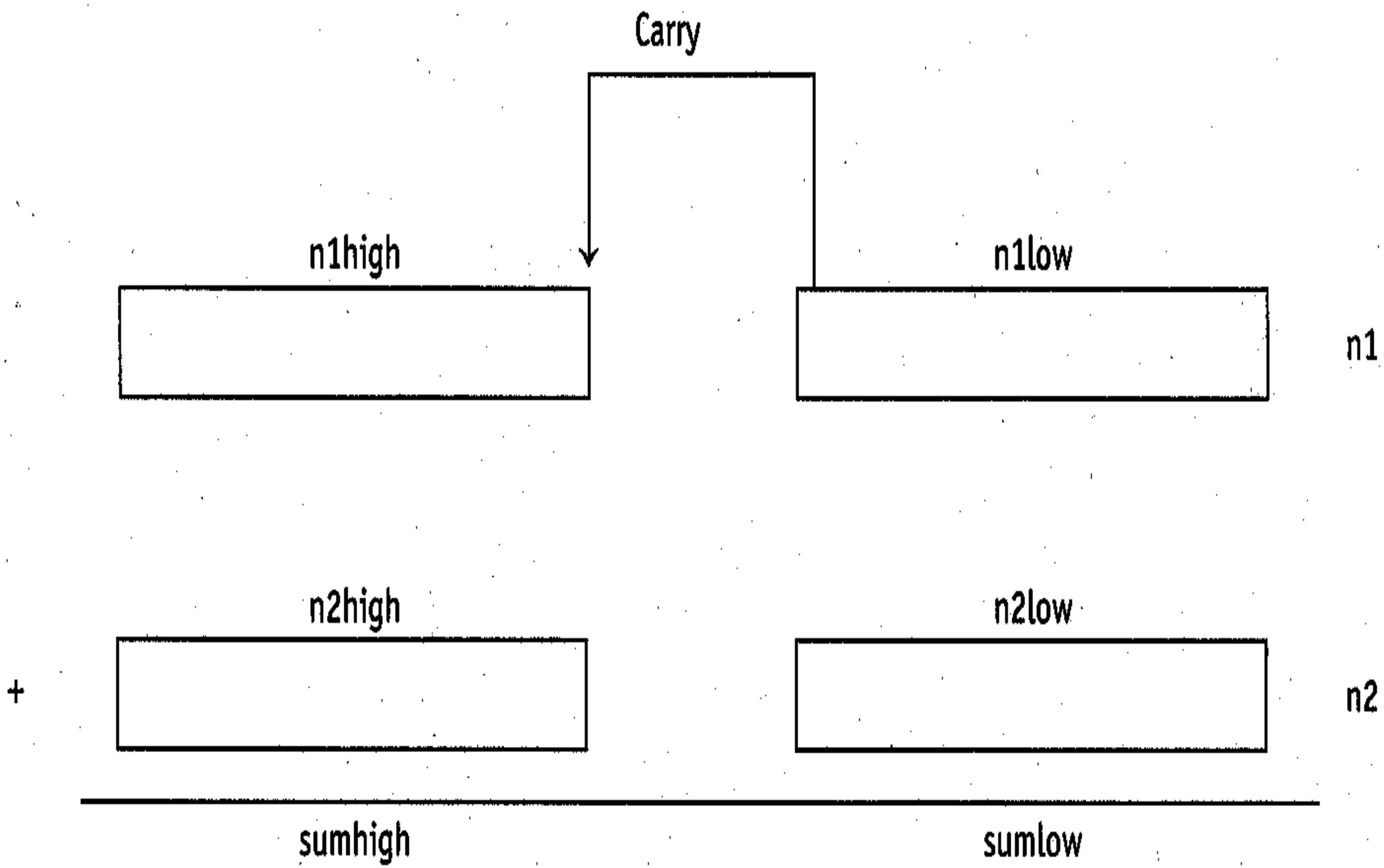
# addy both uses and sets cy

### 12.2.7 addy (Add with Carry)

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| FF7 | addy | Add with carry | ac = mem[sp++] + ac + cy;<br>cy = carry; |

# Multi-word addition

FIGURE 12.4



# Code that performs multi-word addition

```
ld    n1low
add   n2low   ; add low words, carry is saved in cy
st    suml    ; save low sum
ld    n1high
push          ; get high word onto stack
ld    n2high  ; get other high word into ac register
addy          ; add high words and cy
st    sumh    ; save high sum
```

### 12.2.8 `or` (Bitwise Inclusive Or), `xor` (Bitwise Exclusive Or), `and` (Bitwise And), and `flip` (Bitwise Complement)

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| FF8 | or | Bitwise or | ac = ac \| mem[sp++]; |
| FF9 | xor | Bitwise excl or | ac = ac ^ mem[sp++]; |
| FFA | and | Bitwise and | ac = ac & mem[sp++]; |
| FFB | flip | Bitwise complement | ac = ~ac; |

# How addy determines the carry out

**FIGURE 12.5**

| (a) | (b) | (c) | (d) |
|---|---|---|---|
| 0________ | 1________ | 0________ | 0________ n1low |
| 0________ | 1________ | 1________ | 1________ n2low |
| | | 1________ | 0________ sumlow |
| no carry out | carry out | no carry out | carry out |

# Bit-level operations

**FIGURE 12.6**

| | | | | | |
|---|---|---|---|---|---|
| . . . | 1 | 1 | 0 | 0 | first number |
| . . . | 1 | 0 | 1 | 0 | second number |
| . . . | | | | | result |

**FIGURE 12.7**

or

| | | | | |
|---|---|---|---|---|
| . . . | 1 | 1 | 0 | 0 |
| . . . | 1 | 0 | 1 | 0 |
| . . . | 1 | 1 | 1 | 0 |

xor

| | | | | |
|---|---|---|---|---|
| . . . | 1 | 1 | 0 | 0 |
| . . . | 1 | 0 | 1 | 0 |
| . . . | 0 | 1 | 1 | 0 |

and

| | | | | |
|---|---|---|---|---|
| . . . | 1 | 1 | 0 | 0 |
| . . . | 1 | 0 | 1 | 0 |
| . . . | 1 | 0 | 0 | 0 |

To *set* (i.e., make 1) bit 2, we use

```
ld    mask4
push
ld    x
or                ; set bit 2 to 1
st    x
```

where **mask4** is defined with

```
mask4:dw     4
```

To *reset* (i.e., make 0) bit 2, we use

```
ld    maskfffb
push
ld    x
and              ; reset bit 2 to 0
st    x
```

where **`maskfffb`** is defined with

```
maskfffb:  dw   fffbh
```

To *flip* bit 2, we use

```
ld    mask4
push
ld    x
xor          ; flip bit 2
st    x
```

To *test* bit 2, we use

```
ld    mask4
push
ld    x
and               ; zero all bits except bit 2
jz    is_zero
ja    is_one
```

# cali calls function whose address is in the ac register

## 12.2.9 cali (Call Indirect)

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| FFC | cali | Call indirect | mem[--sp] = pc; pc = ac12; |

# The old way of calling a function whose address is in the ac register

```
add  @call  ; adds call 0 instruction to address in ac st
st   * + 1  ; stores call instruction over next instruction
dw   0
```

where **@call** is defined with

```
@call:  call 0
```

New way of calling a function whose address is in the ac register:

cali

# sect and dect are good for count-controlled loops

### 12.2.10 sect (Set Ct) and dect (Decrement Ct)

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| FFD | sect | Set ct register | ct = ac; |
| FFE | dect | Decrement ct reg | if (--ct == 0) pc++; |

# A count-controlled loop that iterates 100 times

```
        ldc  100
        sect           ; set ct register to 100
start:                 ; start of loop
         .
         .             ; body of loop
         .
        dect           ; decrement ct reg; skip next inst if ct == 0
        ja   start
```

# sodd tests lsb in ac register (jzop and jn test the msb).

## 12.2.11 sodd (Skip on Odd)

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| FFF0 | sodd | Skip on odd | if (ac % 2 == 1) pc++; |

# Sodd skips the next instruction if lsb = 1

```
sodd
ja    even
ja    odd
```

# New bp register

Provides the base address for relative instructions. The bp register does not change during a push, so relative addresses do not change. sp is now used exclusively as a top-of-stack pointer.

# Instructions for new bp register

**12.2.12 esba (Establish Base Address), reba (Restore Base Address), bpbp (bp to bp), pobp (Pop bp), and pbp (Push bp)**

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| FA | esba | Estab base addr | mem[--sp] = bp; bp = sp12; |
| FB | reba | Restore base addr | sp = bp; bp = mem[sp++]; |
| FFF1 | bpbp | Bp to bp | bp = mem[bp]; |
| FFF2 | pobp | Pop bp | bp = mem[sp++]; |
| FFF3 | pbp | Push bp | mem[--sp] = bp; |

Changing relative addresses is no longer a problem because bp does not change on a push.

*p = *p + 1;

```
ldr  2      ; get p
ldi         ; get *p
addc 1      ; add 1 to ac
push        ; this push does NOT increase rel address of p
ldr  2      ; get p (again use relative address 2)
sti
```

# esba saves bp and loads it with new base address (of the called function)

In our shorthand notation, the effect of the `esba` instruction is

```
mem[--sp] = bp;   // save bp by pushing it onto the stack
bp = sp12;        // load bp with value of 12 rightmost bits
                  // of sp (the pointer to the called
                  // function's stack frame)
```

reba deallocates locals and restores bp so that it points to the calling function’s stack frame.

```
sp = bp;          // deallocates local variables
bp = mem[sp++];   // pops stack into bp, restoring bp with address of
                  // the caller's stack frame
```

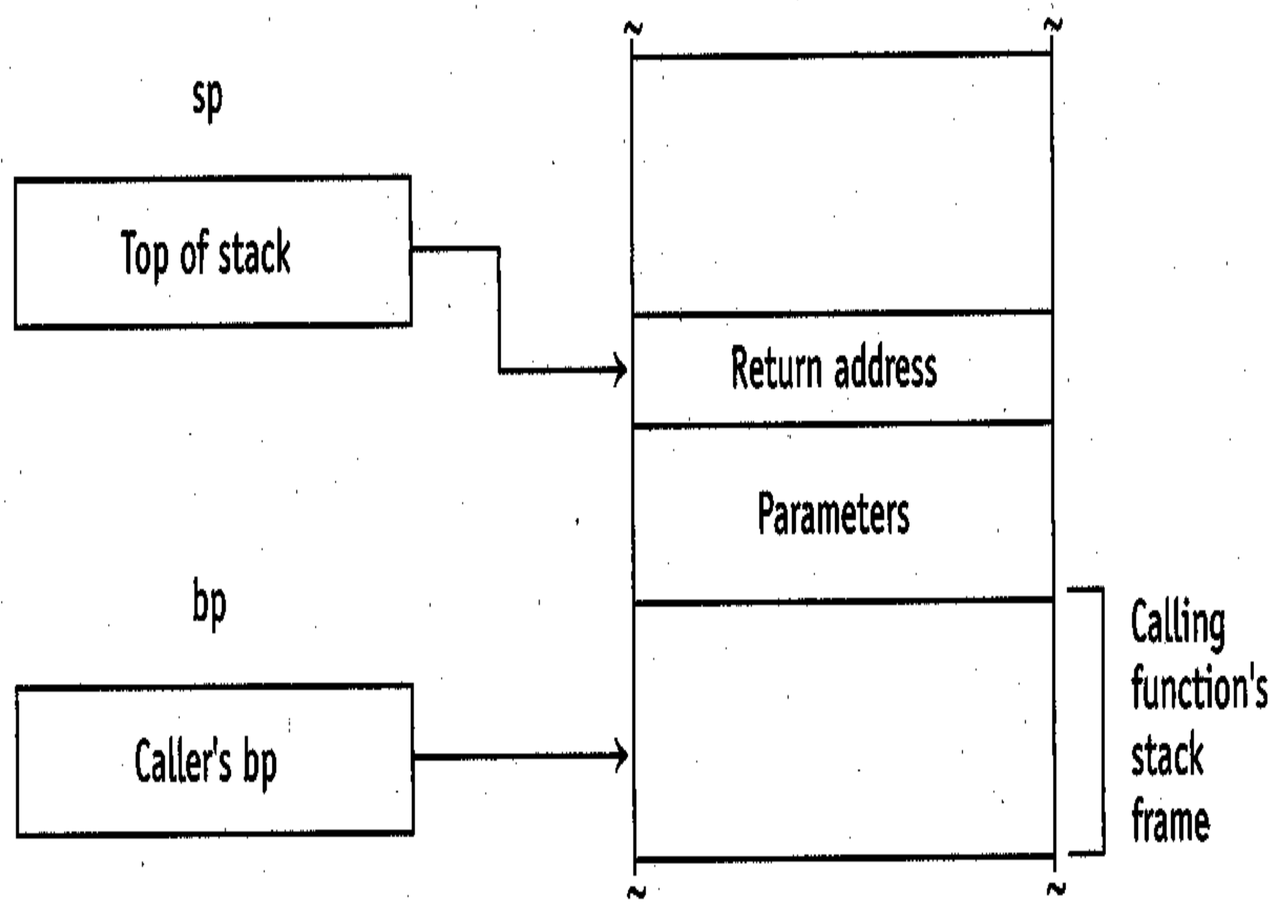
# New function setup.  Note that reba deallocates local variables.

**FIGURE 12.8**

```
f:    esba
        .      <----  allocate local variables here
        .
        .
        .
       reba
       ret
```

# Before esba and after reba

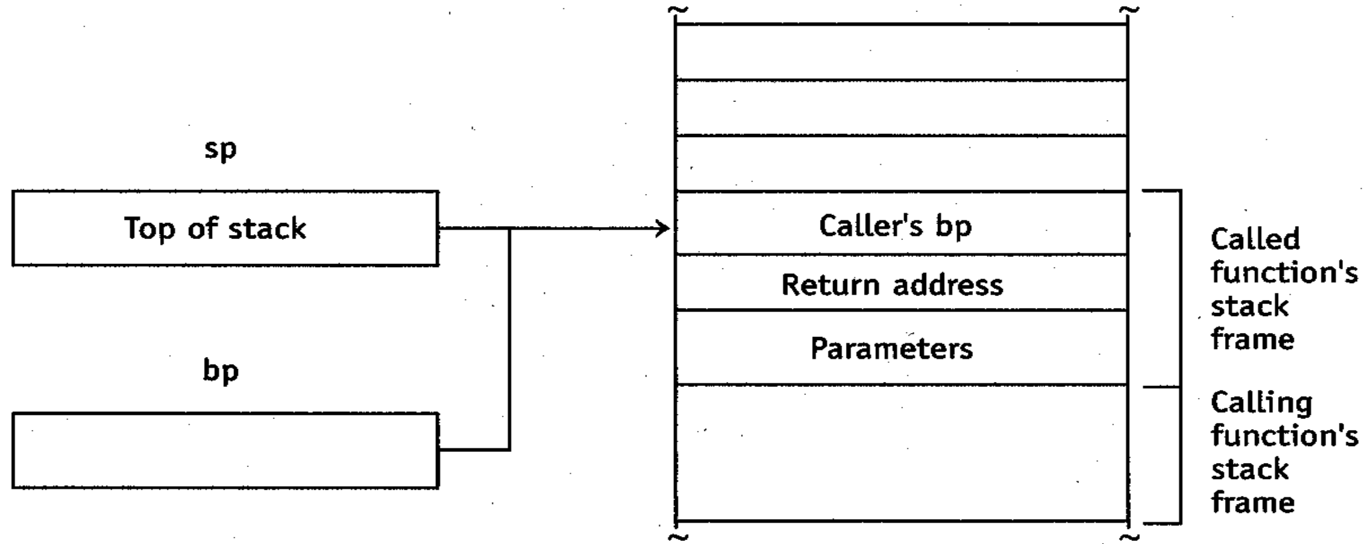
**FIGURE 12.9** a) Before esba



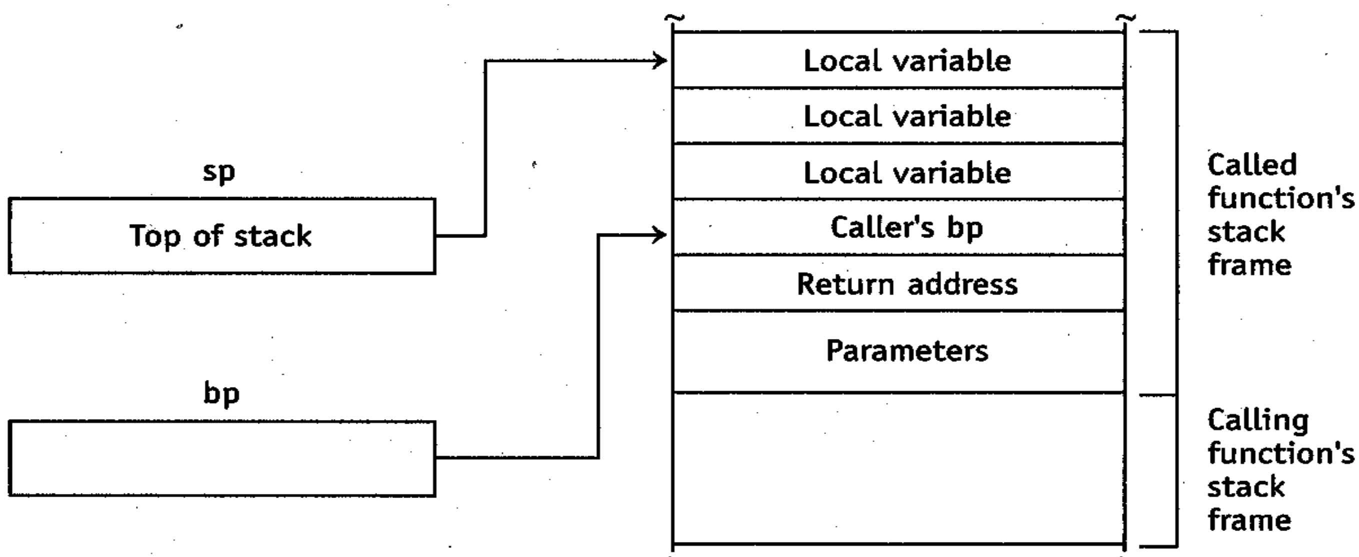

*(continued)*

**FIGURE 12.9** (continued)

b) After esba




c) After allocating local variables



The bp register points to the interior of a stack frame. Thus, relative addresses are positive and negative. See the *preceding* slide.

# The next slides show the assembly code for this program—using the standard and optimal instruction sets.

**FIGURE 12.10** a)

```
void f(int *p, int x)
{
  int y;
  y = *p + x;
  *p = y;
}
```

b)

Standard instruction set

```
            ; int y;
@f$pii:     aloc 1        ; allocate y

            ; y = *p + x;
            ldr  2        ; get p via relative address 2
            ldi           ; get *p
            addr 3        ; add x
            str  0        ; store into y

            ; *p = y;
            ldr  0        ; get y
            push          ; changes sp and relative addresses
            ldr  3        ; get p via relative address 3
            sti           ; store in *p

            dloc 1        ; deallocate y

            ret
            public @f$pii
```

c)

## Optimal instruction set

```
        !o
@f$pii: esba            ; establish base address

        ; int y
        aloc 1

        ; y = *p + x;
        ldr  2          ; get p
        ldi             ; get *p
```

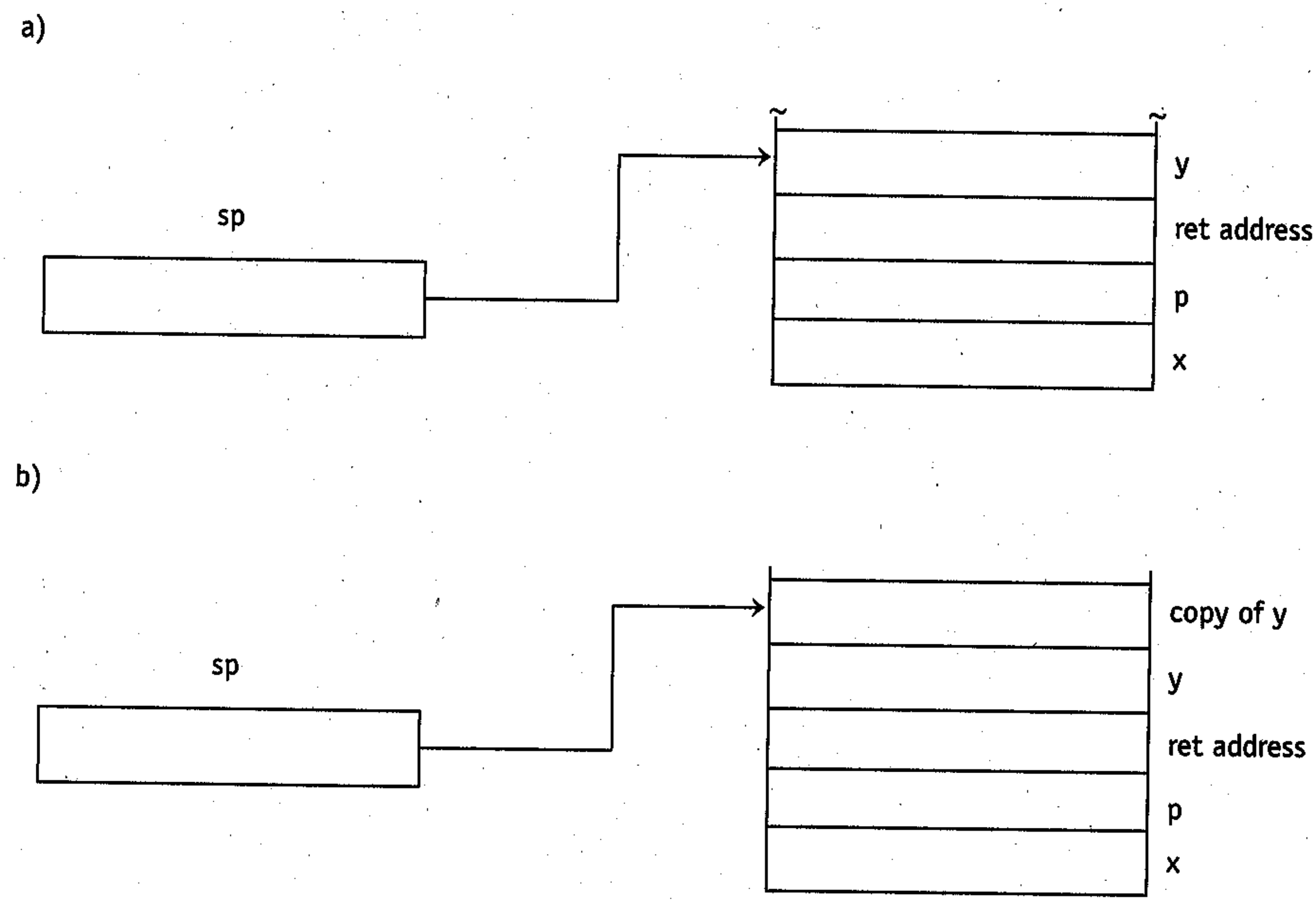
(continued)

**FIGURE 12.10** (continued)

```
          addr 3        ; add x
          str  -1       ; store in y (note negative rel add)


          ; *p = y;
          ldr  -1       ; get y      (note negative rel add)
          push
          ldr  2        ; get p
          sti           ; store in *p


          reba          ; deallocate y and restore bp
          ret           ; return to caller
          public @f$pii
```

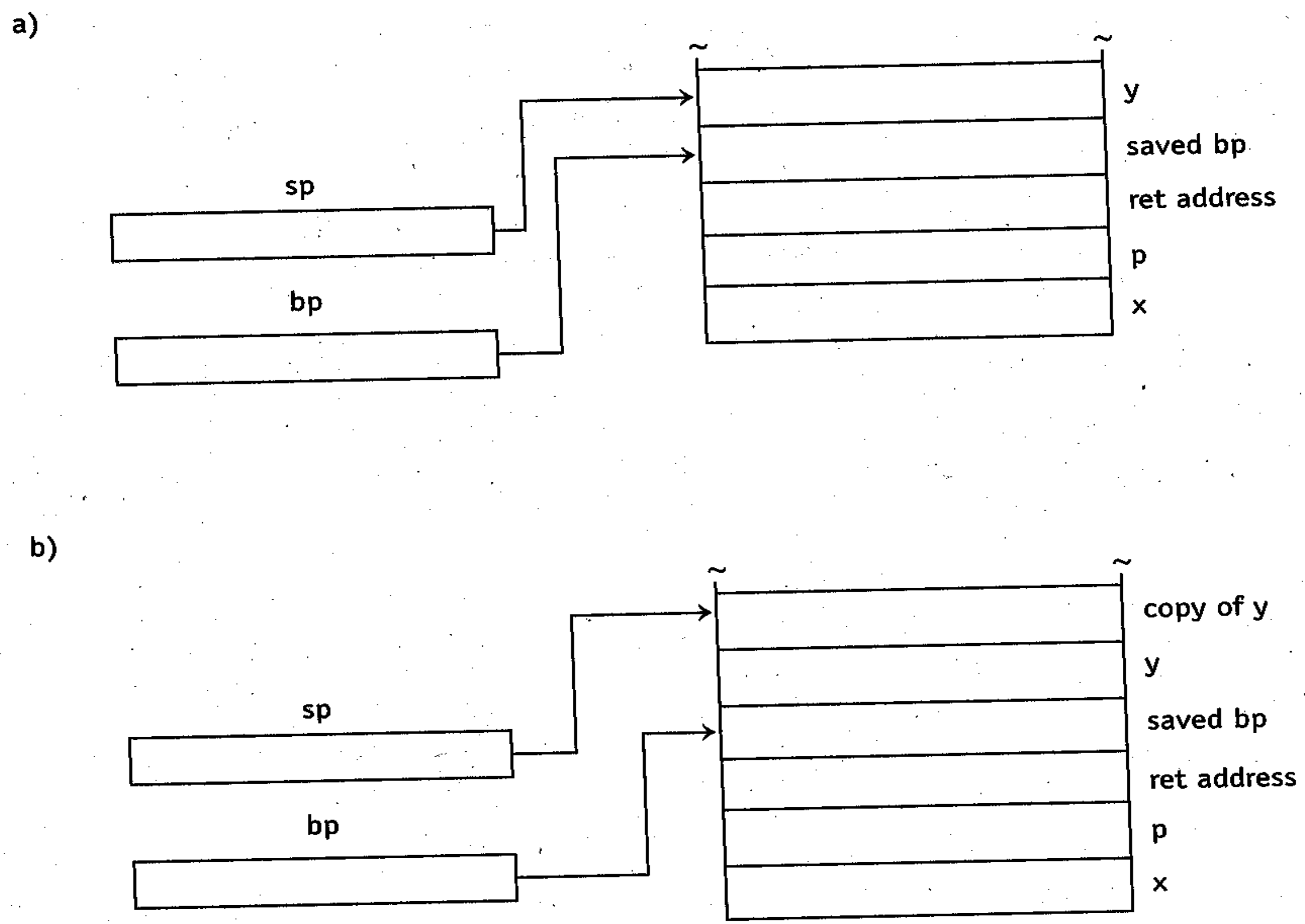
# Changing relative address (2 then 3) of p with standard instruction set

**FIGURE 12.11**



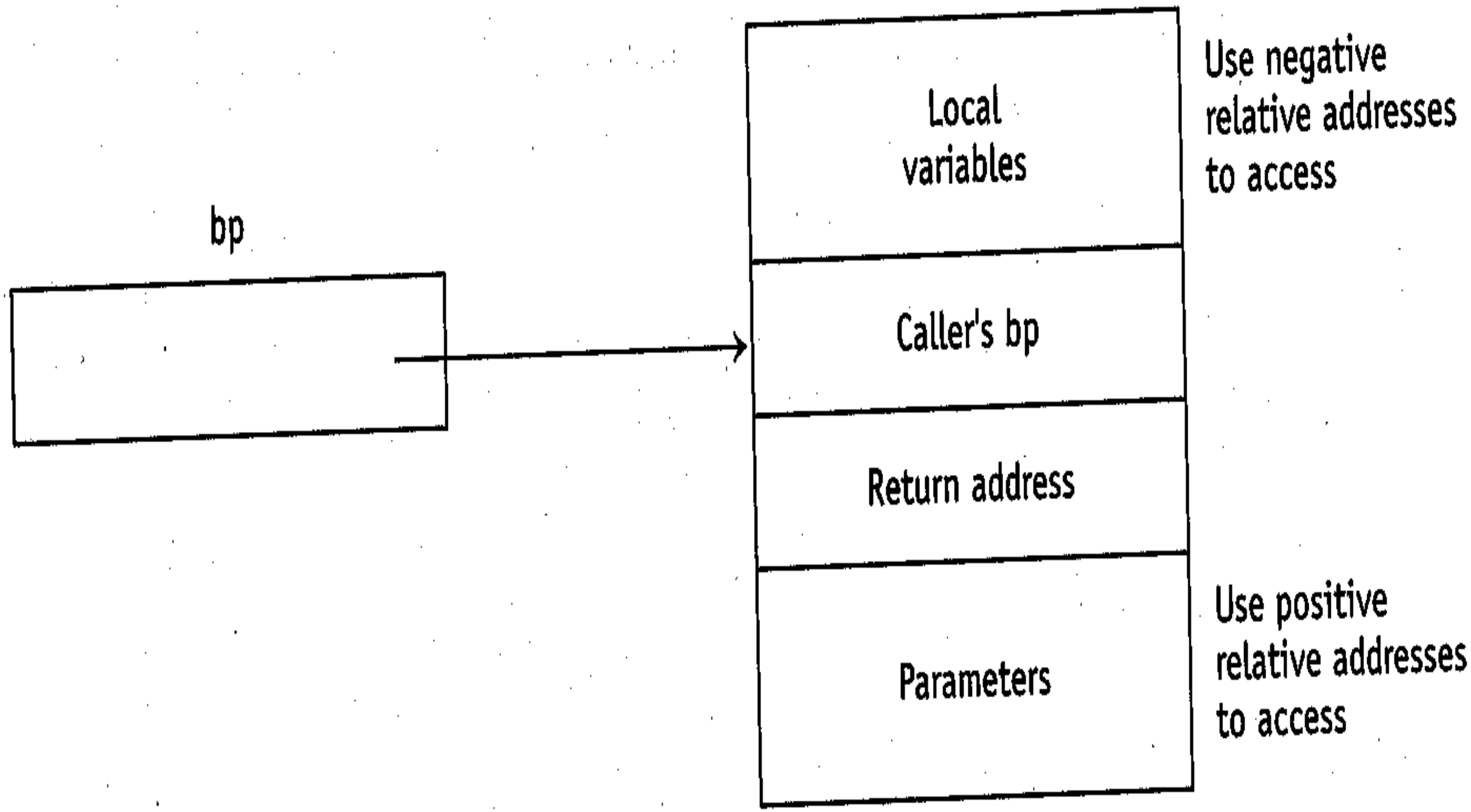
# Constant relative address (2) of p with optimal instruction set

**FIGURE 12.12**



Use negative relative addresses for local variables; positive relative addresses for parameters.

FIGURE 12.13



# New relative instructions

**FIGURE 12.14**

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| 4 | ldr s | Load relative | ac = mem[bp+s]; |
| 5 | str s | Store relative | mem[bp+s] = ac; |
| 6 | addr s | Add relative | ac = ac + mem[bp+s]; cy = carry; |
| 7 | subr s | Subtract relative | ac = ac - mem[bp+s]; |

$-4095 \leq s \leq 4095$

# With cora, we don't need swap-st-swap sequence

## 12.2.13 cora (Convert Relative Address)

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| FC | cora | Convert rel addr | ac = (ac + bp)12; |

# Using cora

```
ldc  3      ; load rel address into ac
cora        ; convert rel address in ac to an absolute address
```

In the standard instruction set, the same conversion requires the awkward sequence

```
swap            ; swap top-of-stack pointer into ac
st   @spsave    ; save top-of-stack pointer
swap            ; restore sp
ldc  3          ; get relative address
add  @spsave    ; add top-of-stack pointer to get abs address
```

# We can use ldc-cora for both positive and negative relative addresses

Instead of using

```
ld    @_2
cora
```

where `@_2` is defined as $-2$ to get the absolute address for relative address $-2$, we can use

```
ldc   -2
cora
```

and avoid having to define the constant $-2$.

# A word-by-word copy occurs at the microlevel when bcpy is executed.

### 12.2.14 bcpy (Block Copy)

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| FFF4 | bcpy | Block copy | while (ct--)<br>mem[ac++] = mem[mem[sp]++];<br>sp = sp + 1; |

# Using bcopy

```
s:      dw    'ABCDE'
d:      dw    '12345'
```

We first initialize the **ct** register to 5:

```
ldc     5      ; get 5
sect           ; set the ct register to 5
```

We then push the source address and load the **ac** register with the destination address:

```
ldc     s      ; get address of s
push           ; push on stack
ldc     d      ; load ac with the address of d
```

We then perform the copy:

```
bcpy
```

which copies the string 'ABCDE' on top of the string '12345'.

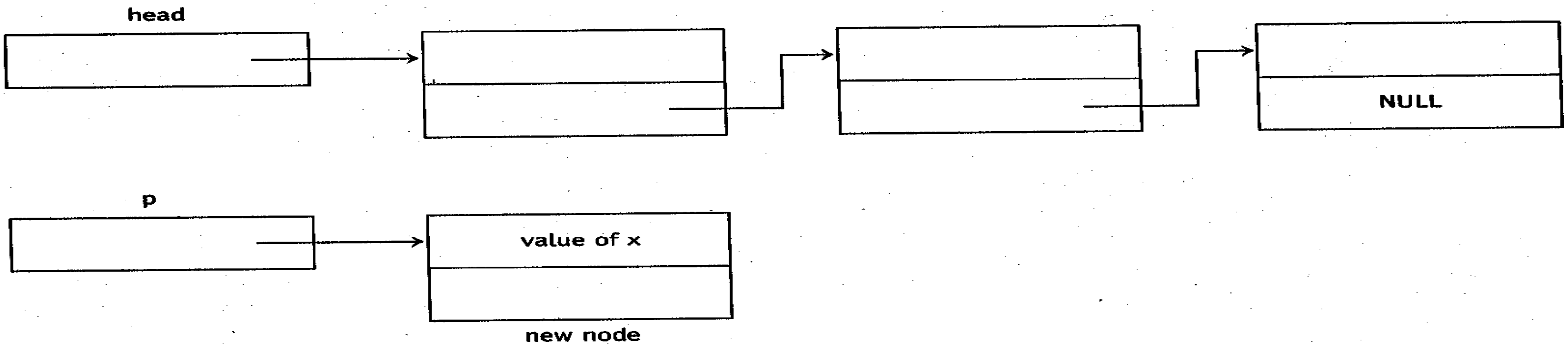
We will illustrate the optimal instruction set with a program that creates and traverses a linked list.

**FIGURE 12.15**

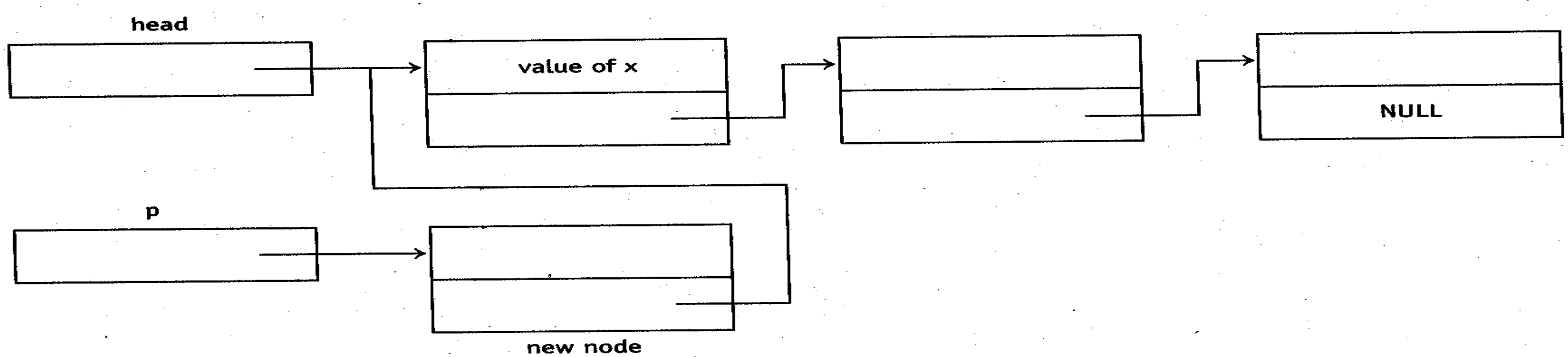
```
#include <iostream>
using namespace std;

struct NODE {
   int data;
   NODE *link;
};
//=========================================================
// traverse displays the data in a linked list in node order
void traverse(NODE *p)
{
   while (p) {
     cout << p -> data << endl;
     p = p -> link;       // move p to next node
   }
}
//=========================================================
// get_data prompts for and inputs an integer
void get_data(int &x)
{
   cout << "enter positive int (or negative int to end)\n";
   cin >> x;
}
//=========================================================
int main()
{
   NODE *head, *p;
   int x;
   head = NULL;
   get_data(x);
   while (x >= 0) {           // end loop on negative number
     p = new NODE;            // allocate new node
     p -> data = x;
     p -> link = head;        // new node pts to head node
     head = p;                // head ptr pts to new node
     get_data(x);
   }
   cout << "Traversing list\n";
   traverse(head);
   return 0;
}
```

**FIGURE 12.16**

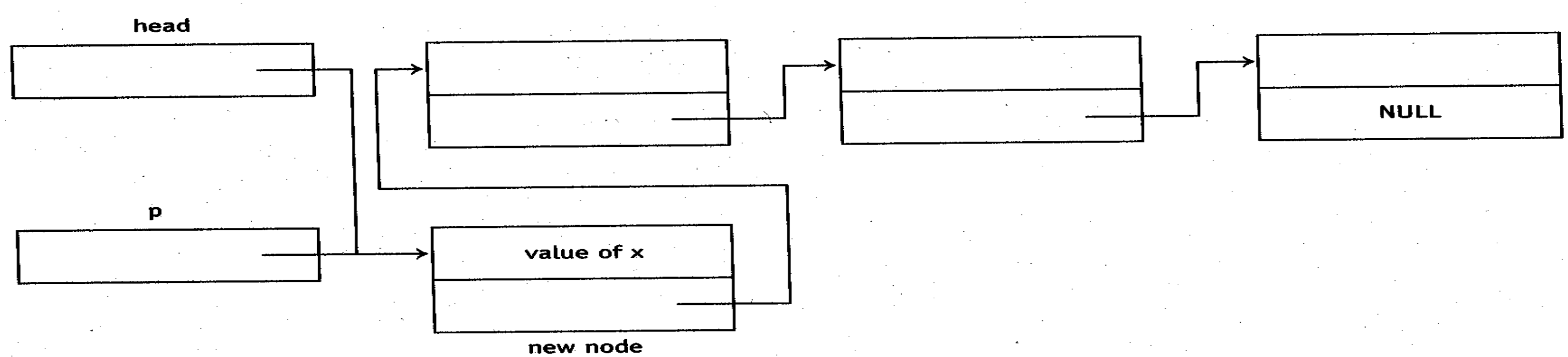
a)




b)




c)



# Putting a new node on the list

1. Get the link field of the new node to point to the node that is currently the head node (see Figure 12.16b). We do this with

```
p -> link = head;
```

2. Get **head** to point to the new node (see Figure 12.16c). We do this with

```
head = p;
```

**FIGURE 12.17**

```
              !o
              ; void traverse(NODE *p)
@traverse$p4NODE:
              esba
@L0:          ldr  2           ; while (p) {
              jz   @L1
              ldr  2           ; cout << p -> data << endl;
              ldi
              dout
              ldc  '\n'
              aout
              ldr  2           ; p = p -> link;
              addc 1
              ldi
              str  2
              ja @L0
@L1:          reba
              ret
;============================================================
@get_data$ri:
              esba
              ldc  @m0         ; cout << "Enter positive int ...\n"
              sout
              din              ; cin >> x;
              push
              ldr  2
              sti
              reba
              ret
;============================================================
main:         esba
              aloc 1           ; NODE *head;
              aloc 1           ; NODE *p;
```

*(continued)*

**FIGURE 12.17**
**(continued)**

```
            aloc 1              ; int x;

            ldc   0             ; head = NULL;
            str -1

            ldc   -3            ; get_data(x);
            cora
            push
            call @get_data$ri
            dloc 1

@L2:        ldr   -3            ; while (x >= 0) {
            jn    @L3

            ld    @avail_ptr ; p = new NODE;
            str   -2
            addc 2
            st    @avail_ptr

            ldr   -3            ; p -> data = x;
            push
            ldr   -2
            sti

            ldr   -1            ; p -> link = head;
            push
            ldr    -2
            addc   1
            sti

            ldr -2              ; head = p;
            str -1

            ldc -3              ; get_data(x);
            cora
            push
            call @get_data$ri
            dloc 1

            ja    @L2

@L3:        ldc @m1             ;  cout << "Traversing list\n";
```

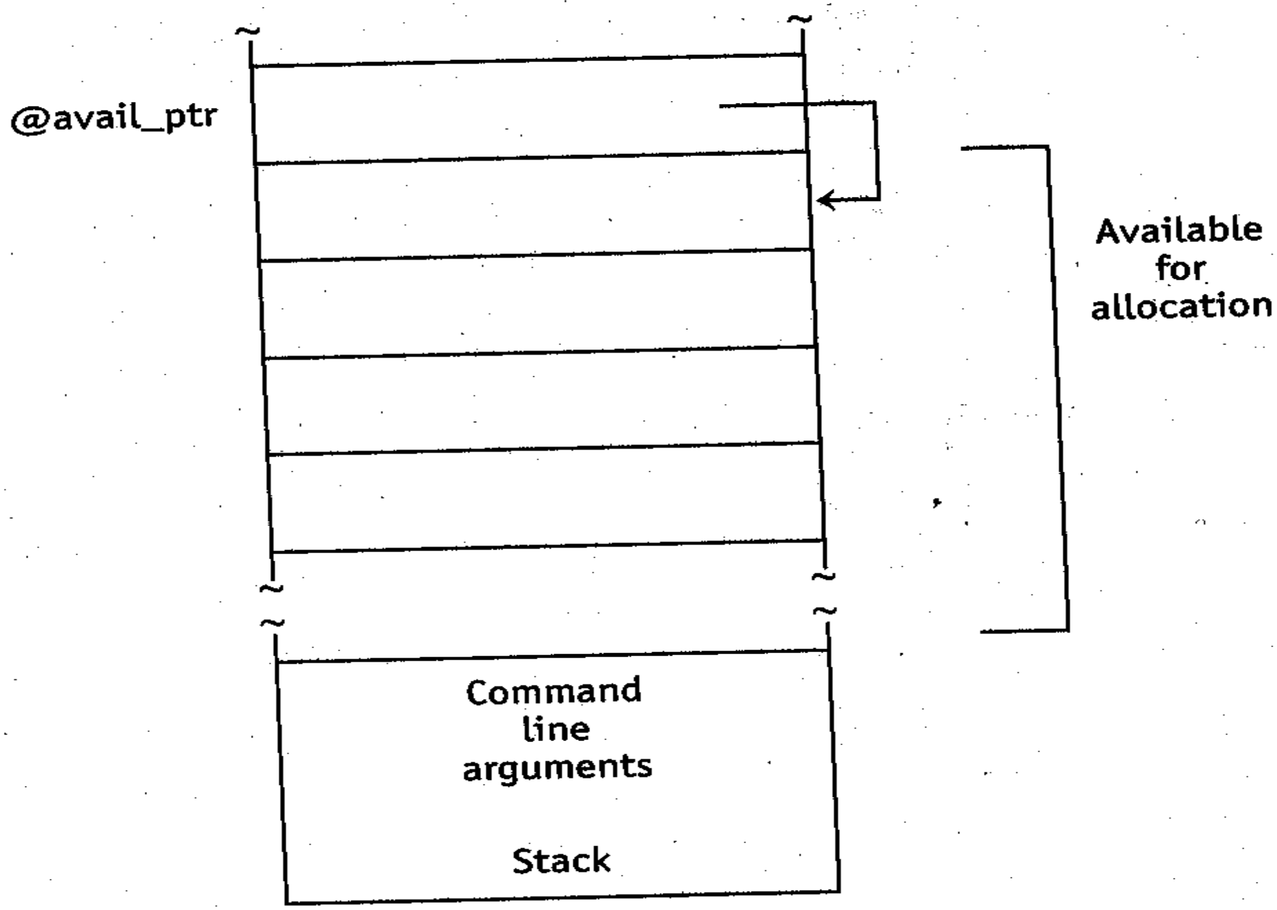
*(continued)*

FIGURE 12.17 (continued)

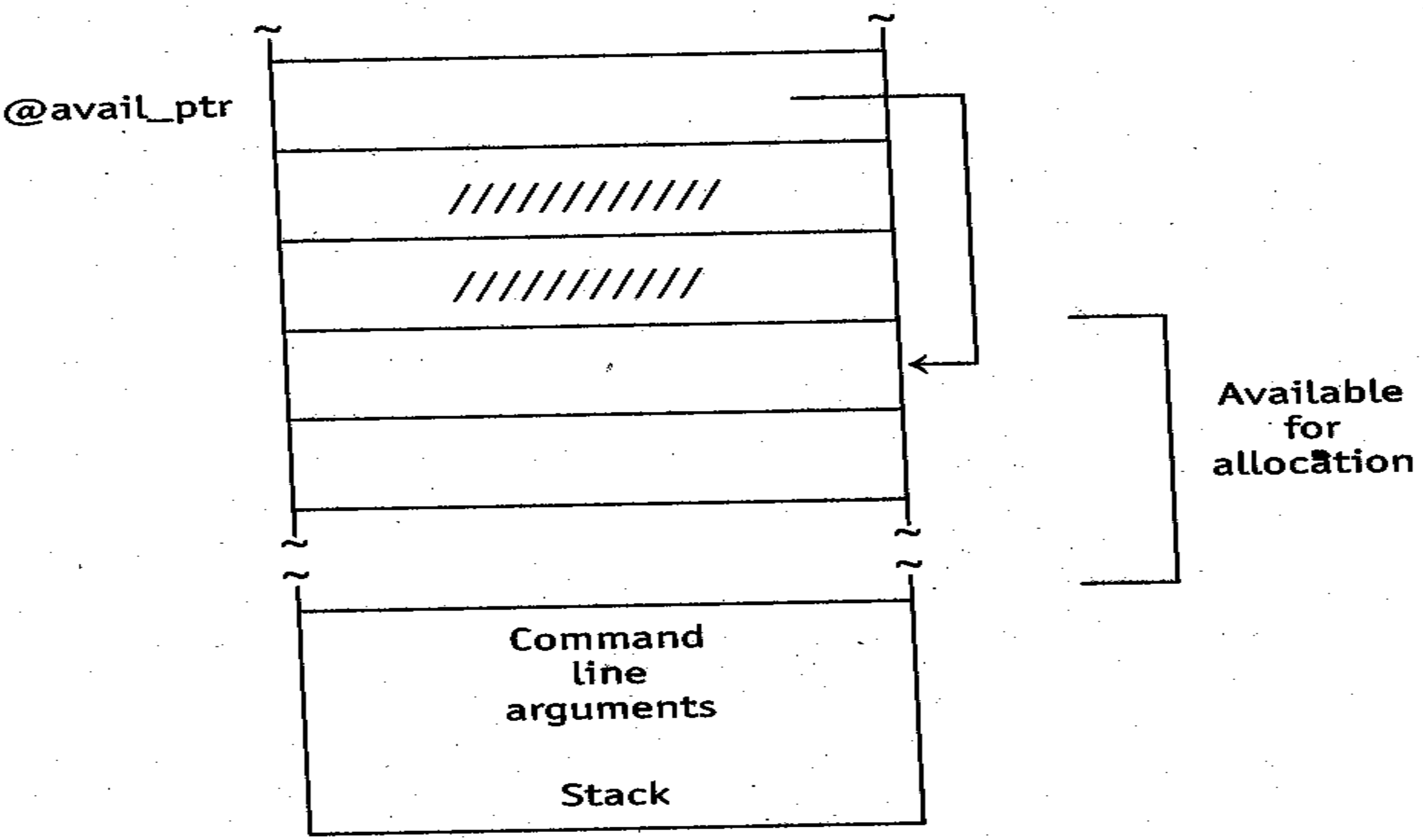
```
            sout

            ldr  -1          ; traverse(head);
            push
            call @traverse$p4NODE
            dloc 1

            ldc  0           ; return 0;
            reba
            ret
;==========================================================
@m0:        dw               "enter positive int (or negative int
                             to end)\n";
@m1:        dw               "Traversing list\n";
            public @traverse$p4NODE
            public @get_data$ri
            public main
@avail_ptr: dw   * + 1
```

# Dynamic memory allocation. Initial configuration

**FIGURE 12.18** a)



# After allocating two words



# Progress report

- Now have immediate instructions addc and subc.
- sect and dect frees up the ac register. Helpful because there are too few accumulator registers (only one) on H1.
- cora does not corrupt the sp register like swap does.  Now it is easy to get the address of an item on the stack.
- Have multiply and divide instructions.

# Progress report continued

- Because of the new bp register, relative addresses do not change.
- Now have bcpy for block copies.
- Now can easily call a function given its address using cali.
- scmp and ucmp perform comparisons correctly.
- Multi-word operations now supported by add, addc, addr, and addy.
- Bit-level operations supported.

# Remaining problems

- Not enough main memory
- Strings stored inefficiently
- No index register
- dloc and aloc limited to + 255, -255